Part 0: Skeleton Code

To read, or not to read, that is the question

In this project you'll be implementing some common join algorithms and a limited version of the Selinger optimizer. We've provided a brief introduction into the new parts of the code base you'll be working with.

For **Part 1** we recommend you read through:

- **common/iterator** - Details on backtracking iterators, which will be needed to implement joins
- **Join Operators** - Details on the base class of the join operators you'll be implementing and some useful helper methods we've provided
- **query/disk** - Details on some useful classes for implementing Grace Hash Join and External Sort

For **Part 2** we recommend you read through:

- **Scan and Special Operators** - These talk about additional operators that you'll use while creating query plans
- **query/QueryPlan.java** - Gives a high level overview of a QueryPlan and some details on how to create and work with them

common/iterator

The `common/iterator` directory contains an interface called a `BacktrackingIterator`. Iterators that implement this will be able to mark a point during iteration, and reset back to that mark. For example, here we have a backtracking iterator that just returns 1, 2, and 3, but can backtrack:

```
BackTrackingIterator<Integer> iter = new BackTrackingIteratorImplementation();
iter.next();     // returns 1
iter.next();     // returns 2
iter.markPrev(); // marks the previously returned value, 2
iter.next();     // returns 3
iter.hasNext();  // returns false
iter.reset();    // reset to the marked value (line 3)
iter.hasNext();  // returns true
iter.next();     // returns 2
iter.markNext(); // mark the value to be returned next, 3
iter.next();     // returns 3
iter.hasNext();  // returns false
iter.reset();    // reset to the marked value (line 11)
iter.hasNext();  // returns true
iter.next();     // returns 3
```

`ArrayBacktrackingIterator` implements this interface. It takes in an array and returns a backtracking iterator over the values in that array.

query/QueryOperator.java

The `query` directory contains what are called query operators. A single query to the database may be expressed as a composition of these operators. All operators extend the `QueryOperator` class and implement the `Iterable<Record>` interface. The scan operators fetch data from a single table. The remaining operators take one or more input operators, transform or combine the input (e.g. projecting away columns, sorting, joining), and return a collection of records.

Join Operators

`JoinOperator.java` is the base class of all the join operators. **Reading this file and understanding the methods given to you can save you a lot of time on Part 1.** It provides methods you may need to deal with tables and the current transaction. You should not be dealing directly with `Table` objects nor `TransactionContext` objects while implementing join

algorithms in Part 1 (aside from passing them into methods that require them). Subclasses of JoinOperator are all located in `query/join`.

Some helper methods you might want to be aware of are located here.

Scan Operators

The scan operators fetch data directly from a table.

- `SequentialScanOperator.java` - Takes a table name provides an iterator over all the records of that table
- `IndexScanOperator.java` - Takes a table name, column name, a PredicateOperator (>, <, <=, >=, =) and a value. The column specified must have an index built on it for this operator to work. If so, the index scan will use take advantage of the index to yield records with columns satisfying the given predicate and value (e.g. `salaries.yearid >= 2000`) efficiently

Special Operators

The remaining operators don't fall into a specific category, but rather perform some specific purpose.

- `SelectOperator.java` - Corresponds to the **σ** operator of relational algebra. This operator takes a column name, a PredicateOperator (>, <, <=, >=, =, !=) and a value. It will only yields records from the source operator for which the predicate is satisfied, for example (`yearid >= 2000`) `ProjectOperator.java` - Corresponds to the **π** operator of relational algebra. This operator takes a list of column names and filters out any columns that weren't listed. Can also compute aggregates, but that is out of scope for this project
- `SortOperator.java` - Yields records from the source operator in sorted order. You'll be implementing this in Part 1

Other Operators

These operators are **out of scope** and directly relevant to the code you'll be writing in this project.

- `MaterializeOperator.java` - Materializes the source operator into a temporary table immediately, and then acts as a sequential scan over the temporary table. Mainly used in testing to control when IOs take place
-

- `GroupByOperator.java` - Out of scope for this project. This operator accepts a column name and yields the records of the source operator but with the records grouped by their value and each separated by a marker record. For example, if the source operator had singleton records `[0,1,2,1,2,0,1]` the group by operator might yield `[0,0,M,1,1,1,M,2,2]` where `M` is a marker record.

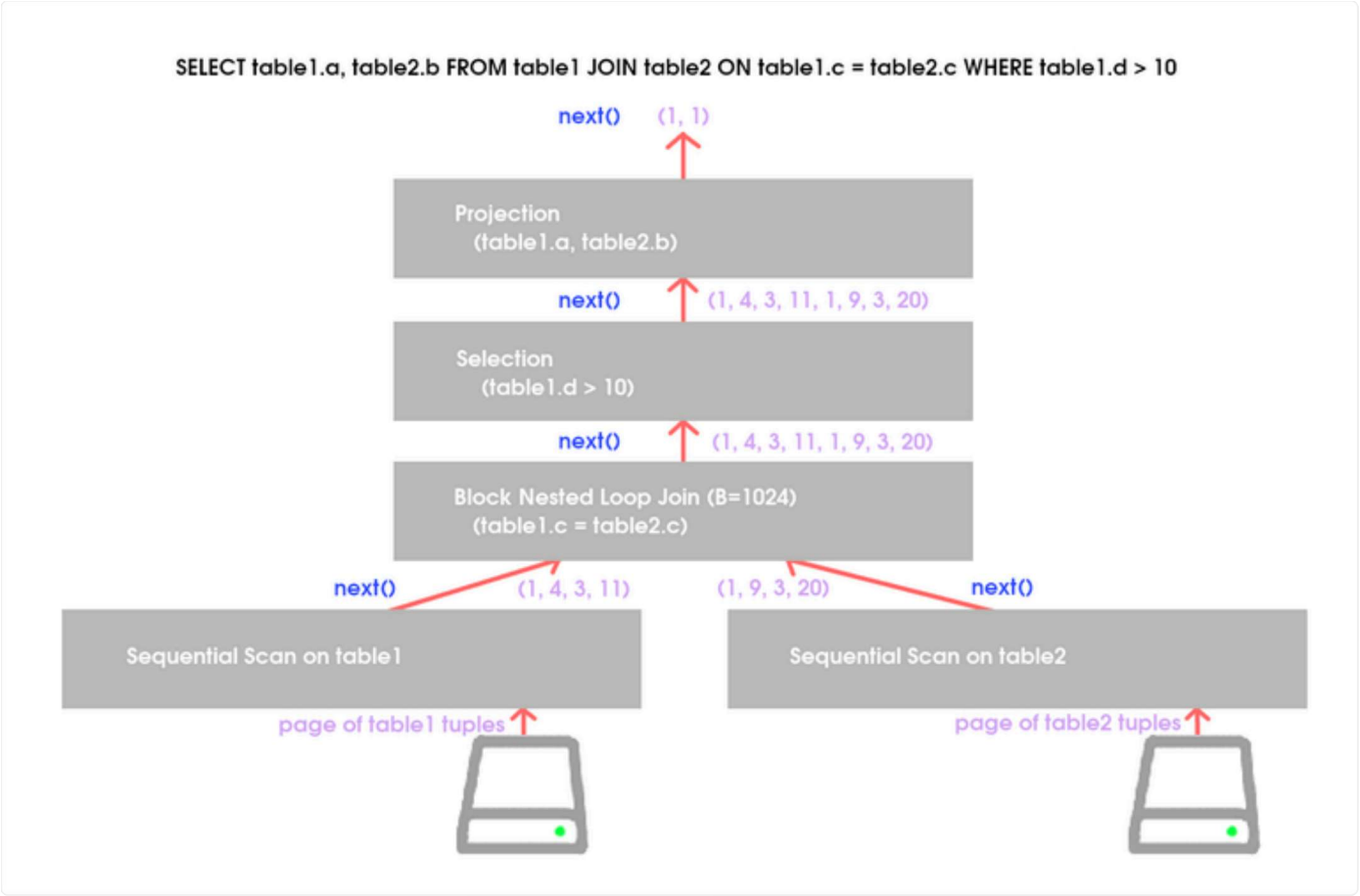
query/disk

The classes in this directory are useful for implementing Grace Hash Join and External Sort, and correpond to the concept of "partitions" and "runs" used in those topics respectively. Both classes have an `add` method that can be used to insert a record into the partition/run. These classes will automatically buffer insertions and reads so that at most one page is needed in memory at a time.

query/aggr

The classes and functions in this directory implement aggregate functions, and are **not** necessary to complete the project (though you're free to browse through them if you're interested).

query/QueryPlan.java



This is the *volcano model*, where the operators are layered atop one another, and each operator requests tuples from the input operator(s) as it needs to generate its next output tuple. Note that each operator only fetches tuples from its input operator(s) as needed, rather than all at once!

A query plan is a composition of query operators, and it describes *how* a query is executed. Recall that SQL is a *declarative* language - the user does not specify *how* a query is run, and only *what* the query should return. Therefore, there are often many possible query plans for a given query.

The `QueryPlan` class represents a query. Users of the database create queries using the public methods (such as `join()`, `select()`, etc.) and then call `execute` to generate a query plan for the query and get back an iterator over the resulting data set (which is *not* fully materialized: the iterator generates each tuple as requested). The current implementation of `execute` simply calls `executeNaive`, which joins tables in the order given; your task in Part 2 will be to generate better query plans.

SelectPredicate

SelectPredicate is a helper class inside of QueryPlan.java that stores information about that selection predicates that the user has applied, for example `someTable.col1 < 186`. A select

predicate has four values that you can access:

- `tableName` and `columnName` specify which column the predicate applies to
- `operator` represents the type of operator being used (for example `<`, `<=`, `>`, etc...)
- `value` is a DataBox containing a constant value that the column should be evaluated against (in the above example, `186` would be the value).

All of the select predicates for the query are stored inside the selectPredicates instance variable.

JoinPredicate

JoinPredicate is a helper class inside of QueryPlan.java that stores information about the conditions on which tables are joined together, for example: `leftTable.leftColumn = rightTable.rightColumn`. All joins in RookieDB are equijoins. JoinPredicates have five values:

- `joinTable`: the name of one of the table's being joined in. Only used for toString()
- `leftTable`: the name of the table on the left side of the equality
- `leftColumn`: the name of the column on the left side of the equality
- `rightTable`: the name of the table on the right side of the equality
- `rightColumn`: The name of the column on the right side of the equality

All of the join predicates for the query are stored inside of the joinPredicates instance variable.

Interface for querying

You should read through the `Database.java` section of the main overview and browse through examples in `src/test/java/edu/berkeley/cs186/database/TestDatabase.java` to familiarize yourself with how queries are written in our database.

After `execute()` has been called on a `QueryPlan` object, you can print the final query plan:

```
Iterator<Record> result = query.execute();
QueryOperator finalOperator = query.getFinalOperator();
System.out.println(finalOperator.toString());
```

```
-> SNLJ on S.sid=E.sid (cost=6)
        -> Seq Scan on S (cost=3)
        -> Seq Scan on E (cost=3)
```

Previous
Getting Started

Next
Part 1: Join Algorithms

Last updated 5 months ago